12

Programming with Shared Bulletin Boards in Asynchronous Distributed Systems

Kenneth P. Birman
Thomas A. Joseph
Pat Stephenson

TR 86-772
August 1986

TECHNICAL REPORT

Department of Computer Science
Cornell University
Ithaca, New York

86 9 15 190

12

Programming with Shared Bulletin Boards in Asynchronous Distributed Systems

Kenneth P. Birman
Thomas A. Joseph
Pat Stephenson

TR 86-772
August 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

REPORT DOCUMENTATION PAGE AD-A171902

Form Approved
OMB No. 0704-0188
Exp Date Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b DECLASSIFICATION/DOWNGRADING SCHEDULE

3 DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Relilease
Distribution Unlimited

4 PERFORMING ORGANIZATION REPORT NUMBER(S)
TR 86-772

5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Kenneth P. Birman, Assist. Prof
CS Dept., Cornell University

6b OFFICE SYMBOL *(If applicable)*

7a NAME OF MONITORING ORGANIZATION
Defense Advanced Research Projects Agency/IPTO

6c. ADDRESS *(City, State, and ZIP Code)*
Computer Science Department
405 Upson Hall
Cornell University, Ithaca, NY 14853

7b. ADDRESS *(City, State, and ZIP Code)*
Defense Advanced Research, Project Agency
Attn: TIO/Admin, 1400 Wilson Blvd.
Arlington, VA 22209

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
DARPA/IPTO

8b OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ARPA Order 5378
Contract MDA-903-85-C-0124

8c. ADDRESS *(City, State, and ZIP Code)*
Defense Advanced Research, Project Agency
Attn: TIO/Admin., 1400 Wilson Blvd.
ARlington, VA 22209

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE *(Include Security Classification)*
Programming with Shared Bulletin Boards in Asynchronous Distributed Systems

12. PERSONAL AUTHOR(S)
Kenneth P. Birman, Thomas A. Joseph and Patrick Stephenson

13a TYPE OF REPORT
Special Technical

13b TIME COVERED
FROM ________ TO ________

14 DATE OF REPORT *(Year, Month, Day)*
August 1986

15 PAGE COUNT
32

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The authors ~~We~~ consider loosely coupled distributed computing systems in which processes interact through shared resources which are modeled as bulletin boards. The first part of the paper formalizes the notion of consistent behavior when unreliable processes concurrently access a bulletin board. The remainder of the paper discusses software techniques for implementing consistent bulletin boards in a network of processors lacking shared memory. Applications for the ~~our~~ approach range from asynchronous interprocess communication to mechanisms for achieving mutual exclusion, deadlock detection and for building distributed database systems.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b TELEPHONE *(Include Area Code)*

22c OFFICE SYMBOL

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

PROGRAMMING WITH SHARED BULLETIN BOARDS IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

Kenneth P. Birman, Thomas A. Joseph, Pat Stephenson

Dept. of Computer Science
Cornell University, Ithaca, New York

ABSTRACT

We consider loosely coupled distributed computing systems in which processes interact through shared resources which are modeled as *bulletin boards*. The first part of the paper formalizes the notion of consistent behavior when unreliable processes concurrently access a bulletin board. The remainder of the paper discusses software techniques for implementing consistent bulletin boards in a network of processors lacking shared memory. Applications for our approach range from asynchronous interprocess communication to mechanisms for achieving mutual exclusion, deadlock detection and for building distributed database systems.

1. Introduction

Two distinct styles of distributed programming have become prevalent in recent years. The *imperative* style is typified by *remote procedure calls* [Birrell] [Cooper]: a tightly coupled, synchronous mechanism through which processes interact. The basic characteristic of this programming style is that one process can force another to execute code on its behalf; if a request cannot be handled promptly there is no alternative but to implement some sort of request queueing or synchronization mechanism. The *advisory* style has received somewhat less attention. According to this approach, each process "publishes" information that eventually becomes accessible to other processes, but without actually forcing them to act on it [Cheriton]. Advisory systems are thus relatively loosely coupled and asynchronous. A good example of the advisory programming style is the *bulletin-board* abstraction found in some artificial intelligence applications: a collection of processes (expert systems) interact by posting problems and relevant data on a common bulletin board; each process checks the board at its own convenience.

In this paper, we focus on the adaptation of data abstractions like the A.I. bulletin board to a setting characterized by processes distributed among multiple sites (lacking shared memory) in a network, communicating with one another in an advisory manner. Henceforth, we refer to these as *bboard* data structures, or just *bboards*. Our work was motivated by the desire to provide a simple, easily used interface to programmers who do not wish to be involved with the low level details of distributed computing. Bboards provide such an interface in a way that interposes minimal overhead.

In any distributed system, process and site failures and recoveries can occur, and other processes that remain operational may need to detect and act on such events. Ensuring that behavior will be correct in the presence of failures is a potentially difficult problem that, if not treated at a system-wide level, can greatly complicate application software. Accordingly, we have integrated into the bboard support a failure detection mechanism that ensures that if any process observes a failure using a bboard, all processes will do so, and moreover that the ordering of failures relative to other events will be the same from the perspective of all observers. As a result a bboard provides a uniform interface that processes can use both to communicate with one another and to monitor one another. As the examples in Section 8 illustrate, problems that are difficult to solve in the absence of this sort of structure are often easy to solve using bboards.

The paper begins by developing a formal model that captures the logical behavior of a bboard executing in the presence of failures and recoveries. We show that different forms of correctness may be desired of a bboard, depending on how it will be used, and that the cost and performance of a given implementation are limited by the desired correctness constraints. The consequences of interacting with multiple bboards having different correctness constraints are also explored, and we are able to show that good performance can often be obtained by placing resources into separate bboards that respect differing correctness constraints.

Next, we discuss a particular bboard implementation. The implementation is flexible in that it allows users to provide their own code to implement operations supported by the bboard. On the other hand, no effort is made to optimize the implementation by taking advantage of the semantics of operations, and bboard operations are not permitted to nest by invoking other bboard operations during execution. The implementation is targeted towards a network that is structured as a set of clusters of sites that communicate over local area networking devices, the clusters being interconnected by long haul connections. We also assume that processes and sites fail by halting, without previously taking any incorrect actions, and that network *partitioning*, whereby subgroups of sites form within which communication is possible, but between which it is severely degraded or impossible, is rare and rapidly resolved.[1] Our implementation is based on a set of reliable multi-

[1] This is because our protocols sometimes block during network partitioning, although no incorrect actions result.

cast protocols which we developed and proved correct under these assumptions in [Birman-b].

The concluding sections of this paper discuss applications of the bboard approach. We describe bboard implementations of binary semaphores, an A.I. bboard of the sort cited above, a distributed deadlock detector, a replicated file supporting transactional access and concurrent updating [Joseph], and a collection of primitives drawn from the Linda S/Net Kernel [Carriero].

2. The model

Our model consists of *processes* lacking shared memory that communicate with one another through bboards. A bboard consists of a set of *objects*. An object may be a simple single-valued variable, or it may be a complex data structure like a queue or a tree. Each type of object has an allowable set of operations. A process interacts with a bboard by invoking an operation on it. (A bboard operation is simply a set of operations on objects on the bboard.) The bboard then accesses the objects and returns a result to the process. The invocation may also cause a change in the state of some of the objects accessed. Operations that do not cause the state of any object to be changed are called *read-only* operations.

A process learns about the behavior of other processes in the system by means of the results returned by the bboard for the operations it invokes. From the values it receives, a process could, in general, deduce that another process has completed a certain action, or that certain events have occurred in a certain order. Since the bboard responds to each process independently, a correct bboard should ensure that its responses are such that the views of the system held by different processes are in agreement with one another. In other words, the bboard should be *consistent*.

Consistency comes in many flavors and not all of them will be suitable for a particular application. The most natural form of consistency is *atomic consistency*: all operations appear globally atomic. This means that if one process observes the outcome of operation a before that of operation b, then all other process will observe them in the same order. This form of consistency is easy and convenient to use; the bboard appears to be a tightly-coupled shared memory.

For some applications, atomic consistency is stronger than is necessary. Consider the example of a simple airline reservation system, consisting of a database (perhaps implemented as a bboard) containing the current bookings, and a set of processes that apply reservations and cancellations to the database. Several passengers make and subsequently cancel reservations for a particular flight. The order in which the processes observe reservations and cancellations of *different* passengers is unimportant. It is necessary, however, that each passenger's cancellation be everywhere observed to be after his reservation. What is important here is that a passenger's reservation is *causally related* to his cancellation. This leads us to a second type of consistency - *causal consistency*. To determine whether one invocation causally affects another requires, in general, knowledge of the semantics of the operations invoked, which may not be available to a bboard. However, if there is no means by which knowledge of an invocation i by a process P could have reached process Q before Q performs i', then i and i' cannot be causally related. All other invocations are *potentially causally related*. This is discussed in detail in [Lamport]. A causally consistent bboard orders all potentially causal invocations, that is, all processes observe the same order for invocations that could causally affect one another, but processes may differ in the order in which they observe the effects of operations that cannot causally affect one another. This form of consistency is useful because it is cheaper to implement than atomic consistency; hence an application that uses atomic consistency where causal consistency would suffice incurs an unnecessary overhead.

Finally, we consider *minimal consistency*, where the orders in which processes observe events may differ arbitrarily. This form of consistency is useful in applications where a process needs to know whether other processes have carried out certain actions, but does not care about the order in which they occurred.

Each form of consistency places restrictions on the order in which processes view events in the system, minimal consistency placing the fewest restrictions, atomic consistency the most. Processes learn of the actions of other processes only by means of the results returned by the bboard. Hence the statement "process P observes event a before event b" should be interpreted as meaning that "the results that the bboard returns to P are in agreement with the view that a occurred before b." In terms of the bboard, then, the consistency requirement only specifies the behavior it should present at its external interface, and does not restrict the order in which actions are taken internally, provided that the results obtained externally are correct. In typical implementations, one would expect a correlation between the order in which actions are taken internally and the results obtained externally. However, it is the flexibility obtained by decoupling the two that leads to efficient runtime behavior, as will be seen in Section 7.

Whereas consistency refers to the order in which different processes view events occurring in the system, synchronization refers to the degree of control that processes have in specifying this order. It may be necessary, for example, for a process that updates the value of a variable to ensure that the new value not be stored until some other process has read the old one. In our model, processes achieve this sort of synchronization using *guards*. When a process invokes an operation o on a bboard, it may label the invocation with a guard, which essentially specifies a set of invocations that must be performed before the execution of o. In terms of results returned by the bboard, this means that the result returned for operation o must reflect the outcomes of the invocations specified by the guard.

Another aspect to be considered is fault-tolerance. A fault-tolerant application requires a means of detecting when a process fails, and a means for different processes to agree on its occurrence. It is often also necessary for different processes to consistently order the failure relative to other events in the system. We use guards for this purpose too. A guard may specify, for example, that a particular operation should be performed only after the failure of a certain process. A bboard ensures that every process observes the same order between a failure event and any other event, regardless of the type of consistency followed by the bboard. In addition, the bboard ensures that every process observes the failure of another only *after* it has observed the effects of all invocations of the failed process, except for invocations whose guard requires it to be ordered after the failure. This would happen, for example, if an asynchronous action is posted to clean up actions that may be only partially completed if a failure occurs.

We also allow the existence of multiple bboards in a system, because it may be useful to have different types of consistency on different variables. Indeed, this will be essential in any large system.

3. The formal model

The *specification* of a bboard defines the semantics of its operations by describing its behavior when operations are invoked on it sequentially. It gives the results that should be returned by the bboard for every totally ordered sequence of operations on it. For example, the specification of a bboard may state that the results to be returned for the invocation **insert**(a, x) [insert item a on queue x], followed by **insert**(b, x), followed by **first**(x) [return the first item on queue x] should be **done**(), **done**() and **done**(a) respectively. It might also state that the results to be returned for the sequence of invocations **insert**(a, x), **first**(x) and **first**(x) should be **done**(), **done**(a) and **empty**() respectively. A specification may be non-deterministic, that is, it may be possible for the same sequence of invocations to return different results. There are many ways in which a specification may be written: using logical predicates, using state machines, or even by exhaustive enumeration; the actual method chosen does not concern us.

Even though the specification of a bboard is given in terms of totally ordered sequences of invocations, in practice a bboard will be invoked concurrently. A single process may issue concurrent invocations; invocations from different processes will in general be concurrent. The relationship between the results returned in such a setting to the specification of a bboard depends on the type of consistency followed by the bboard.

A process operates by invoking operations on a bboard, waiting for their results, doing some computation based on these results, issuing more invocations, and so on. We model a process P by a set of invocations I_p and a partial order $\rightarrow_p$ on these invocations. Each process has a unique name, which we call its *p_name*. An invocation is a tuple (g, o), where g is the guard for the invocation and o is the operation to be performed, with appropriate arguments. Recall that o may consist of operations on more than one object on the bboard. The partial order $\rightarrow_p$ gives the order in which P presents invocations to the bboard. $a \rightarrow_p b$ means that P presents the invocation b to the bboard after receiving the results of invocation a. Thus b is potentially causally dependent o

The behavior of a bboard is modeled by its *history*, which is a set of *events* E_{BB}. Events correspond to the execution of operations by the bboard, and to the termination of processes. In addition, there is a distinguished event *INIT*, which corresponds to the initialization of the bboard. Formally, an event is one of

[*INIT*], [*p_name*, *TERM*], [*p_name*, *guard*, *operation*, *result*]. The bboard assigns each event a unique name, which we call its *e_name*. The notation *event(e_name)* is used to refer to the event named by *e_name*, and *e.e_name*, *e.guard*, *e.p_name*, *e.operation*, and *e.result*, to refer to the name, guard, *p_name*, operation, and result of event *e*, respectively.

3.1. Guards

The guard for an event *e* specifies a set of events whose effects *e* must observe. A guard may be nondeterministic, meaning that the set of events satisfying the guard may not be unique. In this case, at least one such set of events must occur before *e*. Notice that guards cannot directly access the objects in a bboard or otherwise sense the bboard "state". Guards are defined as follows:

	Guard g for event e	**{S | S is a set of events satisfying g}**	**Comment**
(i)	ϕ	{ {INIT} }	g is trivially satisfied
(ii)	*e_name*	{ {*event*(*e_name*)} }	e must observe the effects of *event*(*e_name*)
(iii)	[*p_name*, *op*]	{ {[*p_name*, *op*], [*p_name*, *TERM*]} }	e must observe an invocation of operation *op* by process *p_name* or the termination of *p_name*
(iv)	[*p_name*, *TERM*]	{ {[*p_name*, *TERM*]} }	e must observe the termination of process *p_name*
(v)	*after*(*e_name*)g'	{ {a} | a matches the guard g' and a observes *event*(*e_name*)	e occurs *after* a set of events matching the guard g' all of which occur after *event*(*e_name*)
(vi)	g_1 **and** g_2	{ $S_1 \bigcup S_2$ | S_1 satisfies g_1 and S_2 satisfies g_2 }	Both guards must be satisfied
(vii)	g_1 **or** g_2	{ S | S satisfies g_1 } $\bigcup$ { S | S satisfies g_2 }	Either one of the guards must be satisfied.

Some aspects of the guard syntax need clarification. First, a guard of the form [*p_name*, *op*] is considered to be satisfied if process *p_name* fails. Although this does not distinguish abnormal from normal termination, our implementation provides an alternative way to do so (Section 6). Second, any field of a guard may contain a wildcard, denoted by ϕ in our examples; such a field is ignored while constructing a set of events that match the guard. Finally, observe that notation (ii) delays an invocation until a particular event *e_name* occurs, whereas notation (v) delays it until some *subsequent* events (indicated by g') occur. The former form is used when the *e_name* of the enabling event is already known. The latter form is used when an event e has begun a sequence of actions that will be terminated by a subsequent event e'. Although e'.*e_name* is unknown, a guard of the latter sort could be used provided that a pattern that will match e' can be constructed.

4. Consistency of the bulletin board

The type of consistency defines the relationship between the results returned by a bboard and its specification by requiring that certain orderings between events be observed. Minimal consistency only orders events relative to failures. Causal consistency also requires that potentially causal events be ordered, and is hence a stronger requirement than minimal consistency. Atomic consistency requires that all events be ordered. In addition, we require that an atomically consistent bboard orders potentially causal events according to causality. The three forms of consistency then form an inclusive hierarchy.

What makes an event potentially causally dependent on another? If a process P receives the result of an invocation i before it issues invocation i', then i' could be causally dependent on i. In other words, if $i \rightarrow_P i'$, then i' is potentially causally dependent on i. Further, if the guard for invocation i' causes it to be ordered after invocation i, then i' is potentially causally dependent on i. We formalize this by defining the *potential causality relation* introduced in Sec. 1 as follows. A relation $\rightarrow_C$ on the events in E_{BB} is a potential causality relation if it is closed under transitivity and satisfies the following conditions.

(1) If $a \rightarrow_P b$ for some process P, then $a' \rightarrow_C b'$, where a' and b' are the events in E_{BB} corresponding to the invocations a and b by process P.

(2) For every event e in E_{BB}, there exists a set S satisfying e.*guard* such that for all events e' in S, $e' \rightarrow_C e$.

If guards can contain the names of arbitrary events, it may not be possible to construct an acyclic potential causality relation on the events in the history of a bboard. However, if we assume that when a process invokes an operation, it uses only the names of events whose effects it has observed, then an acyclic potential causality relation can always be constructed.

A *view* for an event $e \in E_{BB}$ under an acyclic potential causality relation $\rightarrow_C$ is a total order that includes all events that e is causally dependent on. Formally, let E_V be a subset of E_{BB} such that if $e' \rightarrow_C e$, then $e' \in E_V$ and if $e \rightarrow_C e'$, then $e' \notin E_V$. E_V thus contains all events upon which e is potentially causally dependent, no event which is potentially causally dependent on e, and may or may not contain events that are not potentially causally related to e. Let $\rightarrow_V$ be a total order on the events in E_V such that if a, $b \in E_V$ and $a \rightarrow_C b$, then $a \rightarrow_V b$, that is, $\rightarrow_V$ includes all the relations in $\rightarrow_C$. The relation $\rightarrow_V$ is then a view for e under $\rightarrow_C$.

A total order on the history of a bboard *meets the specification* of the bboard if it is true that, had the invocations had been presented to the bboard in this order and had the bboard returned the same results as in its history, then the bboard would have been performing according to its specification. In other words, the results returned by the bboard to the processes are indistinguishable from the results returned by a correctly functioning bboard that receives invocations sequentially in the given total order.

We now use these definitions to formalize what it means for a bboard to satisfy each of the three types of consistency. A bboard satisfies atomic consistency if there exists a potential causality relation on its history that can be extended to a total order that meets the specification of the bboard. This is just another way of saying that the results returned by the bboard should be in agreement with the picture that all the events were totally ordered, and that this order agrees with potential causality.

A bboard satisfies causal consistency if there exists a potential causality relation $\rightarrow_C$ such that for every event e in E_{BB} there is a view under $\rightarrow_C$ that meets the specification of the bboard. This means that for every event the results returned are such that there appears to be a total order on all potentially causal events, but since different events can have different views, there need be no global order on the events.

A bboard satisfies minimal consistency if there exists a potential causality relation $\rightarrow_C$ such that for every event e in E_{BB} there is a set of events E containing e with the following property: for all $e' \in E$, $e' \rightarrow_C e$, and the events in E can be extended to a total order that meets the specification of the bboard. This requires the bboard to return results based on some but not all of the potentially causal events, and no ordering requirement is present.

5. Making a consistent cut in a bboard

Misra and Chandy define a *consistent cut* [Chandy] in a distributed system with potential causality relation $\rightarrow_C$ to be a set of events S such that if $e \in S$ then for all $e' \rightarrow_C e$, $e' \in S$ as well. We can then define the events at the *front* of a consistent cut S as the set $\{ e \mid e \in S$ and there exists no $e' \in S$ such that $e' \rightarrow_C e \}$. Events, as defined by Misra and Chandy, refer to single-site operations, wheras a bboard event, as we have defined it, may consist of operations on several objects, and may hence occur at multiple sites. Thus a single bboard event may consist of several single-site sub-events. However, the sub-events comprising an individual bboard event e always form the front of a consistent cut and the events E_V in any view for e form a consistent cut. This is interesting because a number of distributed algorithms based on consistent cuts have been developed, hence these can easily be implemented in the context of bboards. Invocations in these types of bboards also satisfy a *containment* property. Let e and e' be two events occurring in a bboard and let E_V and E'_V be the events in views for e and e'. If $e \rightarrow_C e'$ then $E_V \subseteq E'_V$. Moreover, if the bboard is atomically consistent then there is a view such that even if e and e' are not potentially causally related, it is always true that either $E_V \subseteq E'_V$ or $E'_V \subseteq E_V$.

These properties have intuitive interpretations in terms of the way that time is perceived by the processes using a causally or atomically consistent bboard. Essentially, they say that a bboard operation that accesses multiple objects can be thought of as happening instantaneously according to a logical interpretation of time: every other bboard operation is either before or after such an operation, and these *before* and *after* relations respect causality. Thus, in a causally consistent bboard, if one process takes a snapshot of the bboard state and then communicates on the basis of this with another process, the second process always sees a bboard state subsequent to the one seen by the first. A stronger condition holds in atomic bboards: all snapshots are ordered even if the intersection of the objects they access is empty. These observations lead to valuable simplifications of distributed algorithms that operate on bboards. For example, in the deadlock detection algorithm of Section 8, deadlock detection is done by an operation that computes a snapshot of a set of process states maintained as separate objects in a causally consistent bboard.

6. Consistency levels in advisory systems

We have defined a hierarchy of consistency levels, with minimal consistency at the bottom of the hierarchy and atomic consistency at the top. Each level requires an order to be observed on certain events that the lower levels do not: minimal consistency places no ordering requirements, causal consistency requires that potentially causal events be ordered, and atomic consistency requires that ev[illegible]n-causal events be ordered. The question arises, however, of whether these levels of consistency are equivalent in the following sense. Is it possible for a process accessing a bboard to execute a protocol that will order events not ordered by the bboard, thus obtaining behavior equivalent to that from a bboard with a higher level of consistency? Under the advisory model of computing, this is not possible. In this model, process P can only post its information and request that other processes post a response. There is no guarantee that other processes will actually read the posted information, or that they will respond within a finite time. In fact, since P will in general not be aware even of the number of other processes in the system, it will never know when all processes have responded. Even if it is agreed *a priori* that all processes periodically post information regarding events they have recently observed, the fact that there is no bound on the relative speeds of processes means that P could have to wait indefinitely. We note that if such a bound exists, it is possible for the processes to synchronize their actions and order events independently of the bboard [Cristian], but this is a deviation from the advisory model.

Another question that arises is whether the hierarchy can be extended beyond atomic consistency. Atomic consistency requires all individual events to be ordered. A higher form of consistency could require that groups of events be ordered relative to other groups. For example, it could be required that all events by the same process be ordered in the same way relative to all events by other processes. This is a form of *serializability*: the processes behave like transactions. Carrying the analogy further, a still higher level of consistency might provide an ordering on groups of groups of events, and so forth.

Interestingly, once a process has access to an atomically consistent *helper* bboard, higher levels of consistency can be achieved using it. The idea is to use the helper to share a token. Abstract operations to acquire and release the token can easily be implemented, and since the order in which these are performed is fixed by the bboard and is globally consistent, it can be used to generate an ordering on otherwise unordered events or groups of events. The order selected could also be shared by changing the value of the token when releasing it. We exhibit such a token in Section 8, and also show how it can be used to obtain serializability in a causally consistent bboard. Because there is little reason to believe that an explicit bboard with a higher level of consistency than atomic consistency would be much more efficient or have some other strong advantage over a bboard implemented this way, our model was only carried as far as atomic consistency.

7. Implementation

This section describes an implementation of bboards which we are undertaking at Cornell. We begin with a brief overview of the environment within which this implementation functions. Some pragmatic objectives that were relevant to the internal implementation strategy we adopted are discussed. Finally, the communication primitives on which the implementation is based are presented together with the bboard algorithms and a proof that the implementation achieves the desired forms of consistency.

Our implementation permits a single process to interact with multiple bboards of differing consistency levels. Moreover, although data items and operations are associated with specific bboards, as in the model, guards can include *e_names* drawn from multiple bboards. This feature turns out to be quite useful in developing bboard-based application software. We do not believe it would be difficult to extend our model to capture these possibilities but saw little advantage to be gained by doing so.

7.1. Computing and communications environment

Our work assumes a wide-area network of multiprocessing computers supporting message-based interprocess communication (the protocols take advantage of clustering into groups of sites interconnected by a higher speed local network devices to achieve improved performance). Workstations and individual processes fail by *crashing*: execution ceases (no undetectably incorrect messages are sent first) and the local states of failed processes are irrevocably lost. Later, we will discuss prospects for recovering state information after a failure by periodically saving checkpoints.

7.2. Overview and Language Features

The implementation is designed as a package of library routines which is accessible from the C language under BSD 4.3 UNIX. Each client program will be linked to the bboard package at compile time. At run-time, a client can *create* a new bboard or *enroll* in an existing one. It can then issue operations to the bboard using the following interface:

BBoards and sessions

bb = **bboard**(*b_name,ops,level*)
Enroll in or create a bboard. A unique name for the bboard is given together with a list of operations it will support and an indication of the consistency level. A descriptor to use when operations are issued is returned.

initiate *bb:p_name*
Initiate a new session for the designated bboard and the process with unique name p_name.

terminate *bb:p_name*
Terminate the session. If the same processes wishes to interact further with the bboard it should initiate a new session using a different p_name.

Types

bboard_type
The type of a bboard.

ename_type
The type of an *e_name*.

pname_type
The type of a p_name.

Basic invocation

result := {*guard*} *bb.op(args)*
When the guard is satisfied, invoke the operation on bboard *bb* and store the result in *result*. The object(s) to be accessed are identified in the arguments, which the bboard mechanism does not interpret. In addition to the guard syntax from Sec. 3, the special guard *timeout(secs)* is supported; it is satisfied after *secs* seconds have elapsed. The bboard specifier *bb* may be omitted if the process is only accessing one bboard.

Qualified invocations

evar:: <*basic invocation*>
After performing the invocation, the *e_name* that it was assigned is stored in *evar*.

evar:: **async** <*basic invocation*>
The operation is performed asynchronously. Its *e_name* is stored in *evar* immediately and execution continues concurrently with that of the invocation.

result := **join** *evar*
Execution pauses until the asynchronous invocation to which *evar* corresponds terminates. The result is then stored in *result*. **Join** can only be executed by the process that originated an asynchronous invocation, and provides a way to interrupt a long-running bboard operation, for example at the behest of a user.

cancel *evar*
The designated invocation is cancelled if it has not yet been executed. **Cancel** can only be executed by the process that originated an asynchronous invocation.

Specification of operations

op(*e: ename_type, args*)
The bboard performs an operation by invoking the user-supplied routine **op**, passing it the associated *e_name* and any arguments provided

by the caller.

Other system functions

alive(p_name)	*True* if session *p_name* is still active and *false* otherwise.
failed(p_name)	*True* if process that initiated session *p_name* failed while the session was still active, *false* if not.

In our implementation, a single client can initialize and interact with multiple bboards if desired. We will refer to the processes that are enrolled in a bboard as its *components*. A component invokes guarded operations using the syntax given above and the guard notation defined in Section 3. Our bboard facility is responsible for packaging each invocation into a message, dispatching the message to the components of the bboard, delaying the execution of an invocation until its guard is satisfied, and then invoking the operation with the appropriate arguments. The programmer who implements a bboard provides code for the operations that it supports, in the form of procedures with value/result semantics. In addition, the programmer provides mechanisms for managing storage for bboard data items, procedures to *create* or *initialize* data items, and (if desired) a way to *query* the bboard to learn what objects are currently on the bboard. Because objects may not reside at the same address in different components, objects are normally identified symbolically in the arguments to an operation and mapped to the appropriate address using a symbol table at runtime. A symbol table package is included as part of our bboard implementation.

7.3. Fault tolerance and degree of replication

If a bboard has multiple components and one fails, our implementation is such that the bboard will be left in a consistent state and the failure will be detectable using the guard mechanisms described earlier or the system function *failed*. Fault-tolerance considerations have lead us to replicate all objects on a bboard in every process where the bboard is used. Full replication may not be necessary for some applications, however, and we discuss prospects for replicating bboard objects to lesser degrees at the end of this section.

7.4. Underlying communication primitives

The implementation is based on a set of communication primitives described in [Birman-b], which enable a component to send a message to one or more destination components. The primitives guarantee *atomic delivery* of messages to all destinations. Atomic delivery is usually taken to mean that if the sender does not fail, all operational destinations will receive the message, and that if the sender does fail but any destination receives the message, then all other destinations will receive it too. However, this allows for the possibility that one of a causal sequence of broadcasts will be received at all its destinations, while a broadcast that it depended upon is delivered nowhere -- behavior that we wish to rule out. We therefore modify the standard definition of atomicity as follows. Recall that under our failure model, when a process or site fails all information regarding its current state is lost. It follows that the scenario in which a process receives a message and then fails is indistinguishable from one in which it failed before receiving the message, unless it took an external action like sending a message before failing. Hence, if a process P receives a message m and fails without taking an external action, we do not require that m be delivered to the other destinations. However, if P sends a message m' to Q after receiving m and before failing, then unless Q fails as well, m must be delivered to its remaining destinations. This is because the state of Q may depend on the contents of m.

Our primitives differ in several other respects from what has normally been called *atomic broadcast* [Schneider] [Chang] [Cristian]. First, atomic broadcast protocols provide all or nothing delivery to a static set of processes (often, all processes in the system). In our situation, the set of components of a bboard can change dynamically, hence at the time a primitive is invoked, the set of processes that will ultimately receive the broadcast message is undetermined. Secondly, most atomic broadcast protocols provide a *global message delivery ordering* property in addition to atomic delivery: broadcasts are received in the same order everywhere in the system. To satisfy this property is costly: it requires a multi-phase or token based protocol, or a delay before message delivery can be attempted. Such strong ordering is only needed in atomic bboards, and it is too costly to accept in cases where it isn't actually needed. To overcome this problem, our primitives satisfy varying ordering constraints, and have latency that varies accordingly. Finally, unlike the previously reported work, our communication primitives are integrated with a mechanism for dealing with failure and recovery at the level of individual processes.

7.4.1. The GBCAST primitive

GBCAST (group broadcast) is the most constrained, and costly, of the four primitives. Arguments to *GBCAST* are a message and the symbolic name of a bboard, which is automatically translated into a set of destinations. The *GBCAST* protocol ensures that if any component receives a broadcast b before receiving a *GBCAST* g, then all components receive b before g. This is true regardless of the type of broadcast b. *GBCAST* is used primarily to transmit information about failures and recoveries to operational components of a bboard. When a component fails, the system arranges for a *GBCAST* to be issued to the operational components of the bboard on its behalf, informing them of its failure. This *GBCAST* is delivered after any other broadcasts from the failed component. A new or recovering component wishing to join an existing bboard also uses *GBCAST* to inform the operational components that it has become available.

Because of the way in which *GBCAST* is ordered relative to other broadcasts, each component can maintain a *view* listing the components belonging to a bboard, updating it whenever a *GBCAST* is received. Although views are not updated simultaneously (in real time), all components observe the same sequence of view changes. Moreover, all components receiving a broadcast b (of any type) will have the same value for the view at the time b is received, and can hence take consistent actions in response to b. Intuitively, we wish for the view to represent a ***logical system state*** in which the message was received simultaneously by all operational bboard components.

GBCAST has a stronger atomicity condition than the other primitives: if a component receives a *GBCAST*, then ***even if it fails*** all others will receive it too. With this condition, *GBCAST* provides an inexpensive way to determine the last component(s) that failed, when all components fail. Bboard components simply record each new view on stable storage: a simplified version of the algorithm in [Skeen] can then be executed on these stable views when components start to recover after the failure is resolved. We use this property in connection with the bboard checkpointing mechanism presented at the end of this section.

7.4.2. The ABCAST primitive

The second primitive, *ABCAST* (atomic broadcast), satisfies a slightly weaker ordering constraint. Any two messages transmitted using *ABCAST* will be delivered in the same order at all common destinations. Further, if the same component performs more than one *ABCAST* to overlapping destinations, then the order of delivery at these destinations is the same as the order of initiation of the *ABCAST* protocol. *ABCAST* is implemented using a 2-phase protocol.

7.4.3. The CBCAST primitive

The third primitive, *CBCAST* (causal broadcast), involves less distributed synchronization then *GBCAST* or *ABCAST*. In an asynchronous distributed system with no shared memory, the only way in which an action a can influence an action b is if they both occur in the same process, with the a occurring before b, or if there is a sequence of messages from process to process which could have carried information about action a to the process carrying out action b. This is formalized in the definition of the *information flow relation* below. The *information flow relation* $\rightarrow_I$ is the transitive closure of the following relations.

(1) If a and b are actions by the same component of a bboard, and a occurs before b, then $a \rightarrow_I b$.

(2) If a is the sending of a message by a component, and b is the receipt of the same message by another, then $a \rightarrow_I b$.

If o_1 and o_2 are the actions corresponding to the initiation of two *CBCAST*'s and $o_1 \rightarrow_I o_2$, then the message sent by o_1 is delivered before that sent by o_2 at all common destinations. Note that no delivery order is specified for *CBCAST*'s not related under $\rightarrow_I$. *CBCAST* is implemented using an inexpensive 1-phase protocol that employs piggybacking to enforce this delivery constraint.

7.4.4. The MCAST primitive

The fourth and last primitive is *MCAST* (multicast). *MCAST*(*msg*, *gname*) atomically delivers *msg* to each operational member of *gname*. The delivery order is unconstrained.

7.5. Basic Bboard Implementation

Within the above framework, implementation of the bboard package is straightforward. Each bboard is created with a single component. When a process wishes to enroll in a pre-existing bboard, *GBCAST* is used to broadcast its intention. On reception of an enrollment *GBCAST*, a *coordinator-cohort* algorithm, is used to transfer the bboard state to the new component. In such an algorithm, one bboard component is in charge of the state transfer and the others back it up; one restarts the state transfer should the coordinator fail [Birman-a] [Birman-b]. Since *GBCAST* is totally ordered with respect to other broadcast events, all components have received the same messages and hence are in equivalent states when the state transfer takes place. *GBCAST* is also used to inform bboard components when a component fails, and they use this information when evaluating guards.

When a component of the bboard is presented with an invocation, the following occurs. An *e_name* is generated for the invocation. Next, the information corresponding to the operation to perform, the arguments, the guard, and the generated *e_name* are packaged into a message and sent to all components (including the one that issued the invocation). We denote the sending of such a message for an invocation i as *send*(i). The primitive used to send the message depends on the consistency level of the bboard: *ABCAST* is used for atomic consistency, *CBCAST* for causal consistency, and *MCAST* for minimal consistency.[2] The caller then blocks until the operation is executed as described below and it receives the result, except in the case of an invocation issued with the **async** option, in which case the caller resumes execution immediately.

When a message is delivered to a component (an action we denote by *recv*(i)), the message is added to a *wait queue*, which preserves the order in which messages are delivered. Messages in the wait queue of a component are processed as follows. Starting at the head of the queue (the earliest delivered message), the guard is evaluated to see whether operations have been executed on the local copy of the bboard that satisfy the guard. An expression of the form [*p_name*, *op*] is also considered to be satisfied if a *GBCAST* relating to a failure of process *p_name* has been received. If the guard is true, the operation in the message is executed by invoking the appropriate procedure with the given arguments. If the invocation was issued by the local component, the result of the execution is returned to it, otherwise the result is ignored. The message is then removed from the queue. If the guard is not satisfied, the next message in the queue is examined. Each time an operation is executed, the guards for previously examined invocations may become satisfied, hence the wait queue is reexamined from its head.

In our initial bboard implementation, all bboard components will save *e_names* until the process that issued the event terminates. Then, *e_names* generated by the terminated process are discarded, although its termination status (whether or not it crashed) is saved indefinitely. This approach is simple and should entail low overhead, provided that individual processes do not execute huge numbers of bboard operations. Possible

[2] We allow the user to substitute other protocols for *MCAST* in order to benefit from the bboard interface while also satisfying other application-specific requirements, such as real-time constraints, that our existing protocols would not address.

optimizations are discussed below.

Finally, the *cancel* operation is transmitted using *GBCAST*. This means that all components have received the same set of invocations at the time a *cancel* request is received. Hence, unless the invocation has already been performed everywhere, it is cancelled at all the components.

7.5.1. Performance considerations

This approach to implementing bboards causes all operations to be executed by all components, which may seem to be a costly approach to replicating data. However, a response is needed only from the component local to the process that issued the operation, because all bboard components compute consistent results. Thus, in contrast to other systems that use this approach to replication [Cooper], the caller does not wait until remote bboard components have processed the operation before continuing, and a bboard will normally execute as asynchronously as is possible given the consistency constraints it must respect. A **flush** primitive is also provided to allow a process to block explicitly until all the operations it has initiated have been delivered to remote bboard components. The remaining question is to determine whether the *overall* cost of computation using this method exceeds what might be achieved with some other method. For example, our previous work on *resilient objects* used a coordinator-cohort method in which each operation is executed by a single component (not always the same one), which then distributes the result to the remote components [Birman-a] [Joseph]. One might wonder if such a strategy would tend to give better performance.

In fact, there are several reasons for believing that replicated processing will be beneficial in a bboard implementation. First, the method we have adopted is extremely simple, and this is obviously an advantage. Perhaps more important is that the cost of simply getting a request to remote bboard components will typically be much higher than the cost of processing it. Moreover, the communication primitives are implemented to employ extensive piggybacking when the system load rises. Thus, when the system is under a moderate load, each arriving message is likely to contain multiple requests. Assuming that I/O and scheduling overhead dominate compute time for typical requests, efficiency rises with increasing load. In fact, performance studies of the *ISIS* system, which uses the same primitives described here, confirmed this effect and showed that it can have a dramatic impact on system performance. In essence, the cost of this class of distributed computations, in which execution is relatively asynchronous and the messages describing operations are "small", is best measured by the number of process scheduling events and I/O operations needed to perform the operation -- not the compute time associated with the operations themselves.

7.6. Correctness

Each component executes operations sequentially in the order the corresponding messages are removed from the wait-queue. We assume that there are no errors in the (user-supplied) definitions of objects and operations on them and hence the results returned at each component are in accordance with the specification of the bboard for the total order followed at that component. We show that even though the order in which operations are executed may differ from component to component, the execution yields the desired level of consistency.

7.6.1. Atomic consistency

For atomic consistency, we must show that the order in which operations are executed is the same at all components, and that this order forms a potential causality relation. Because *ABCAST* is used to transmit messages, messages are delivered and added to the wait-queue at each component in the same order. Now each component uses the same deterministic algorithm to remove messages from the wait-queue; hence messages are removed from the wait-queue in the same order at every component. As a result operations are executed in the same order at every component.

For the (total) order in which operations are executed to be a potential causality relation it must satisfy two conditions:

(1) If $i_1 \rightarrow_P i_2$ for some process P, then i_1 must be executed before i_2 at all components.

(2) Every invocation must be preceeded by the execution of a set of invocations that satisfy its guard.

The implementation obviously satisfies condition (2). As for condition (1), note that if $i_1 \rightarrow_P i_2$, this means that the invocation i_2 was presented to the component P after the results of invocation i_1 were known there. Clearly, i_1 was executed at P before i_2. Since all components execute invocations in the same order, it follows that if $i_1 \rightarrow_P i_2$ for any P, then the invocations are executed in this order at all components.

7.6.2. Causal consistency

We must show that the order in which operations are executed at each component is a potential causality relation (that is, it satisfies conditions (1) and (2) above), even though the order may differ from component to component. As above, the condition on guards is obviously satisfied. If $i_1 \rightarrow_P i_2$, then (the message corresponding to) i_1 was removed from the wait-queue at P before i_2 was issued. That is, $recv(i_1)$ occurred at P before $send(i_2)$. In terms of the information flow relation, $recv(i_1) \rightarrow_I send(i_2)$. Since $send(i_1) \rightarrow_I recv(i_1)$ (by definition), it follows that $send(i_1) \rightarrow_I send(i_2)$. Furthermore, any invocation i' that caused the guard of i_1 to be satisfied must also have been executed at P before i_1 and hence before $send(i_2)$. This means that $recv(i')$ occurs at P before $send(i_2)$. Hence $recv(i') \rightarrow_I send(i_2)$. It follows that $send(i') \rightarrow_I send(i_2)$.

We have shown above that if $i_1 \rightarrow_P i_2$, then $send(i_1) \rightarrow_I send(i_2)$ and $send(i') \rightarrow_I send(i_2)$ for any i' satisfying the guard of i_1 at P. Since *CBCAST* is used to transmit information in a causally consistent bboard, it follows that $recv(i_1)$ occurs before $recv(i_2)$ and $recv(i')$ also occurs before $recv(i_2)$ at all components. In other words, the messages corresponding to i_1 and all the invocations that satisfied its guard at P are added to the wait-queues of all components before the message corresponding to i_2. Hence, i_1 will be executed before i_2 at all components.

7.6.3. Minimal consistency

In a minimally consistent bboard, messages about invocations are transmitted using *MCAST*, which observes no order. This makes it difficult to talk about the correctness of a minimally consistent bboard, since even causally related events are unordered. For example, a process might issue an invocation **set**(x, 0) followed by **add**(x, 10) on a bboard and then obtain the result 10 from an invocation of **read**(x). At another component the operations may occur in reversed order, giving a result of 0 for the read. For this reason, we do not envision the use of minimal causality in bboards maintaining objects that have a "state". However, certain real-time systems have behavior that is conveniently modeled by minimal causality. For example, if a sensor generates a high rate of timestamped readings of a device, the timestamp ordering can be used to decide if a particular reading is valid, and it may not be necessary for all readings to be registered provided that a reasonable degree of currency is maintained. In such a setting a minimally consistent bboard provides a simple, uniform, and inexpensive interface that will simplify the software development task.

In the case where a user specifies some alternative protocol to be used in place of *MCAST*, the consistency achieved will depend on the characteristics of the protocol employed. Any detailed treatment of this problem thus becomes the responsibility of the user.

7.7. Optimizations

Several types of optimizations are expected to be valuable for obtaining good bboard performance. First, in the case of read-only operations, it is not necessary to broadcast the invocation to all bboard components, provided that the guard for the event satisfies a minor restriction. The restriction is that the guard not refer to some other read-only event, and we believe it is minor because such synchronization seems to serve no practical purpose. Since a read-only operation will not change the bboard state, and its result is needed only at the component where the invocation occurred, such an operation can be placed directly on the wait-queue at the local site, and correctness will not be compromised. However, the issue now arises of how guard satisfaction can be determined in the case where the read-only event is referenced in the guard of some other (non read-only) event e'. Since e' is not read-only, however, it will be broadcast to all bboard components including the one where e was executed. That component will discover that e satisfies some part of the guard for e'. Rather than satisfying the guard locally, it broadcasts the *e_name* for e to all bboard components, including itself, using the broadcast type appropriate to the bboard. On receiving a message containing *e_names* for read only operations, all recipients apply the *e_name* to their guards. Since all do this, guard evaluations are consistent and the information flow relation is preserved.

A second optimization concerns the partial replication of bboard data. In general, this is a difficult open problem. An important special case arises when the events which satisfy a guard will occur at a superset of the components where a data item resides. In such cases, invocations of operations on a data item need be broadcast only to the components maintaining the item and the implementation will still function correctly. On the other hand, if some event e satisfying the guard for an invocation i is not broadcast to all components where i is to be executed, some will execute i whereas others will leave i waiting. Such behavior could clearly lead to inconsistencies, and a mechanism to resolve this problem is beyond the scope of this paper.

A third possible optimization would allow a process to cache subsequences of the *e_names* generated by other bboard components, while keeping the complete sequence of its own *e_names*. An interrogation mechanism could then be used to inquire about *e_names*, operating much like the mechanism used with read-only operations that we described above. Our initial bboard implementation will not support this optimization.

7.8. Checkpointing the bboard state

It is well known that if a checkpoint/rollback algorithm is used in a nondeterministic system, and the system enters a state in which messages that were sent prior to rolling back are received *after* rollback, even the weak consistency constraints imposed by minimal consistency can be violated (the message may not be reissued in the state that results after rollback). In [Koo] a multi-phase protocol that avoids such behavior is described. Checkpointing a bboard is much more straightforward because *GBCAST* is atomic and totally ordered relative to other sorts of bboard events. To establish a checkpoint, a *GBCAST* is issued to invoke a checkpointing operation in each component, which causes a checkpoint and a timestamp to be written (atomically) to stable storage. If all the components of a bboard fail, components that recover run an algorithm to determine the last ones that failed; when these have recovered, they compare the timestamps of their last checkpoints. The component(s) possessing copies of the most recent checkpoint restart from it. All others then re-enroll in the bboard.

8. Applications

The bboard paradigm is broadly applicable. For example, the A.I. bboard discussed in the introduction might be implemented as a causally consistent bboard with operations to post and read problems and data. This would guarantee that if an expert process starts working on a problem, it will also find relevant data and previously posted solutions to relevant subproblems. Similarly, existing advisory communication primitives can easily be recast into the bboard framework. For example, the Linda S/Net kernel provides four primitives, *IN*, *OUT*, *READ* and *EVAL*, to manipulate a collection of tuples comprising a shared memory [Carriero]. *IN* adds a tuple to the tuple space. *OUT* finds a tuple that matches some pattern and removes it from the tuple space. *READ* performs the same operation, but without removing the tuple. *EVAL* adds an unevaluated tuple whose evaluation begins as soon as the tuple enters the tuple space. These operations could easily be implemented in an atomically consistent bboard.

Below, we show how three well known problems can be solved using bboards. A token passing example demonstrates the use of bboards to achieve fair, efficient mutual exclusion on a shared resource. A deadlock detector illustrates how bboards might be used to maintain a non-trivial distributed data structure; a deadlock check will discover deadlock if and only if one is really present. Finally, a bboard implementation of a transactional replicated file shows how bboards could be used to implement a database system. Because the bboard interface is simple and lightweight, these implementations all should perform well.

8.1. Token passing

A distributed token can easily be implemented using a bbboard. In the implememtation we describe below, a process attempting to acquire the token is given it immediately if the token is free. If the token is in use, processes waiting for it queue up and compete to acquire the token after it is released.

The token is represented by a record containing a field *holder* that stores the *e_name* of the event that caused the token to be acquired by the current holder (ϕ if there is no holder). Any additional fields needed by the application can be added to the token structure. The operations on the bboard are *grab* and *free*, and are modeled after the usual implementation of semaphores using atomic instructions [Peterson]. *Grab* is used while acquiring the token. If the token is free, invoking *grab* causes the *holder* field to be set to the *e_name* corresponding to the invocation. If the token is in use, *grab* does nothing, and the caller deduces that it must wait by examining the *p_name* of the *holder* field after it returns. *Free* is used when releasing the token, and sets the value of the *holder* field to ϕ.

A process wishing to acquire or release a token does so using the interface routines *acquire* and *release*, which in turn invoke the bboard operations described above. The correctness of the solution follows from the fact that invocations on the bboard are totally ordered; hence not more than one process acquires the token at a time. Notice how the guard is used to avoid busy waiting when an attempt is made to acquire the token while it is held by some other process. If the first attempt to grab the token fails, each iteration of the while loop delays the next grab() operation until the process that holds the token (token.holder.p_name) has either

```
-- Interface procedures (used to issue requests to the bboard) --
var token : token_type;

procedure acquire()
begin
  grab();
  while token.holder.p_name # my_pname
    { after(token.holder)[token.holder.p_name, free] } grab();
end acquire;

procedure release()
begin
  free();
end release;
```

failed or released it; either event being subsequent to the acquisition event (token.holder). The solution is slightly inefficient in that every process issues a *grab()* operation each time a *free()* is done; design of a more efficient solution (for example, one that maintains a queue of waiting processes so that a process only issues a *grab()* if it is the "next" holder of the token) is left as an exercise.

8.2. Deadlock detection

Deadlock detection is an example of a non-trivial problem that has an elegant solution when expressed in terms of our bboard approach. Consider the *RPC deadlock detection* problem. A collection of processes are

```
-- The actual bboard operations --
function grab(e : event)
begin
  if token.holder = $\phi$ or not alive(token.holder.p_name) then
    token.holder := e;
end grab;

procedure free(e : event);
begin
  token.holder := $\phi$;
end free;
```

interacung by remote procedure call. A process that has issued such a call waits for the destination process to reply Periodically, a process that has been waiting for a while checks for deadlock; if deadlock is detected and the process has the lowest *p_name* among processes with which it is deadlocked, it cancels its request. The solution is inexpensive because process states are stored in a causally consistent bboard, which can delay transmission to updates to take maximum advantage of piggybacking.

To solve this problem using a bboard, we first introduce a data structure to represent wait-for relationships. A *wait-for* edge is said to exist between P and Q if P cannot execute until it receives a message from Q. The *wait-for digraph* G consists of a set of nodes corresponding to the processes, with wait-for edges inserted according to the above rule. A deadlock exists if and only if a cycle is present in the wait-for digraph. We assume that a process can only execute one request at a time.

The deadlock detector bboard will be a causally consistent bboard supporting the following operations: *insert* an edge, *delete* an edge, and a read only operation to *check* for deadlock. We will assume that an edge is inserted by a process that will have to issue an RPC to some other process before it can reply to a caller. The edge is subsequently deleted by the same process that inserted it prior to sending the reply. No interface is required; these operations are all performed directly on the bboard. Notice that the bboard passes the event name as an argument to each invocation, even though these names were not needed in this example.

To prove that this bboard is correct. we must establish that if a deadlock occurs it will eventually be detectable and that if a deadlock is detected, it corresponds to a wait-for cycle in the real system. Because the bboard is distributed, it is not immediate that these properties hold: many deadlock detection algorithms tend to find *phantom* deadlocks, which result when wait-for edges from different stages of a computation are assembled into a single, inconsistent snapshot, representing a system state that never occurred [Gray]. For example, Q might at some time wait for P, and P may now be waiting for Q. If the old wait-for edge representing Q waiting for P is included into G, a phantom deadlock would be discovered between P and Q and P might abort itself unnecessarily.

The correctness proof is as follows. Assume that the bboard subsystem is live and that a deadlock occurs. Since deadlock is a stable property (unless a process detects the deadlock and aborts, the deadlock per-

```
**procedure** *insert*(e : event; p, q : processid);
**begin**
  $G_q := G_q + (p, q)$;
**end** *insert*;

**procedure** *delete*(e : event; p, q : processid);
  $G_q := G_q - (p, q)$;
**end** *delete*;

**procedure** *check_for_deadlock*(e : event, p : processid): **readonly**;
**begin**
  $G := \bigcup G_q$;
  **if** $P$ is contained in some cycle in $G$ **then**
    **return** true;
  **else**
    **return** false;
**end** *check*;
```

sists), then eventually all copies of the bboard will exhibit a system history in which the wait-for edges representing the deadlock are included. Thus, if a deadlock occurs, it will eventually be detected. Notice that more than one process may detect the deadlock at once.

Now, assume that a deadlock is detected by process P. Then, P has discovered a cycle $P \rightarrow Q_1 \rightarrow \cdots \rightarrow Q_n \rightarrow P$ in G. Consider the portion of this graph consisting of $P \rightarrow Q_1 \rightarrow Q_2$. It cannot be the case that Q_1 responded to P before it waited on Q_2. Otherwise let e be the event whereby Q_1 deletes the edge $P \rightarrow Q_1$ and e' be the event whereby it subsequently inserts the edge $Q_1 \rightarrow Q_2$. We thus have $e \rightarrow_{Q_1} e'$. Hence, $e \rightarrow_C e'$, implying that e would have been applied to G before e'. Similarly, it must be the case that Q_1 is waiting for Q_2, etc. It follows that the deadlock is real.

8.3. Serializable access to concurrently updated data items

Using an atomic bboard together with a causally consistent bboard, a transactional mechanism supporting asynchronous updates to a replicated database can be implemented. Each transaction consists of a *begin* operation followed by sequence of *read* and *write* operations terminated by a *commit* or *abort*, with the usual semantics. A transaction that fails before committing is automatically aborted. Two-phase locking is used to achieve serializability and a write-ahead log to implement abort [Gray].

The approach is as follows. An atomically consistent bboard, denoted *LOCKS*, stores a set of lock variables. These are acquired just like the tokens of the previous example, but are released by the commit or abort of the transaction that holds the lock (for brevity, only the "interface" code is given below). A causally consistent bboard, denoted DB, stores the log and database items. The *begin* operation posts an asynchronous "cleanup" operation; it will be described shortly. *Read* returns the current value of a variable. *Write* first logs the old version of the data item being updated and then performs the update. Because the log record is written before the update is done, the semantics of a write-ahead log are achieved: regardless of how asynchronously the update is done log records are always written before the corresponding update is done. Finally, *commit* logs a commit record and then terminates the session, while *abort* just terminates the session. Termination enables the cleanup operation, which checks to see if the transaction committed (termination due to a failure is treated as an abort). If not, it rewrites the old values of any variables that have been changed. Then it deletes any log records written by the terminated process. Completion of the cleanup operation, in turn, enables the release of any locks acquired by the transaction; locking is thus two-phase. Moreover, since this establishes a causal chain between the termination of a process and any subsequent process that acquires a lock from it, subsequent processes will observe the updates that have been done even if these are very asynchronous (recall that *CBCAST* is used in this case). A formal treatment of this type of causal chaining is given in [Joseph].

The code for the interface used to communicate with the DB bboard and the bboard operations themselves is given below. As in the case of the token, the interface procedures are not really part of the bboards, but rather are used to communicate with them in a stylized fashion. We omit the detailed management of the log data structure, which is implemented by routines log_write, log_delete, restore_from_log and not_logged. We also use a "pointer" notation to pass references to data items, although in practice this would be replaced with a symbolic addressing mechanism.

The implementation needs some discussion. First, examine the asynchronous guarded operations that are issued for log cleanup and lock release. For each session, *begin* creates an asynchronous log cleanup operation

```
--- Transactional operations: interface ---

procedure START()                              -- post a cleanup operation, note its e_name
begin
  termevent:: async { [my_pname, TERM] } log_cleanup(my_pname);
end START;

procedure LOCK(x : data_item)
begin
  ACQUIRE(termevent, x.lock);                  -- see below.
end LOCK;

procedure READ(x : data_item)
begin
        return read(x);
end READ;

procedure WRITE(x : data_item; value: data)
begin
  log_append("write", "x", x.value);           -- log old value
  write(x,value);                              -- update x
end WRITE;

procedure COMMIT()
begin
   log_append("commit", "p", my_pname);        -- log commit record
   terminate my_pname;                         -- end session.
end COMMIT;

procedure ABORT()
begin
  terminate my_pname;                  -- just end session.
end ABORT;
```

```
--- The LOG part of the bboard --
procedure append(e : event, rtype :readorwrite, item : data_item, value : data)
begin                                          -- log commit requests and first write request
  if rtype = "commit" or (rtype = "write" and not_logged("write", item)) then
     log_write(rtype, item, value);
end append;

procedure cleanup(e : event, p_name : processid)
  if not_logged("commit", p_name) then         -- committed?
    restore_from_log(p_name);                          -- no, roll-back DB
  delete_log_records(p_name);                          -- done
end cleanup
```

```
-- The DB part of the bboard --
procedure readonly read(e : event, x : data_item)
begin
  return x.data;                              -- just return the value
end read;

procedure write(e : event, x : data_item, value : data)
begin
  x.data := value;                            -- just set the value
end write;
```

```
-- The interface to the LOCK bboard --
procedure ACQUIRE(termevent : event, lock : lock_type)
var temp : lock_type;
begin
  -- Post an asynchronous operation to release the lock after commit/abort.
  async { termevent } free(lock)
  acqure(lock.mutex);
end ACQUIRE;
```

that waits until the session ends. It does this using a guard will not be enabled until the invoking process terminates. The *e_name* of the cleanup operation is noted in the variable *termevent*. Later, when locks are acquired, they post an asynchronous release operation guarded by {termevent}. That is, *after* the cleanup, lock release events become enabled. These release any locks that the process held prior to ending the session. The locking algorithm itself can be based on the token passing example, and is omitted for brevity. Recall that an invocation with a guard that becomes enabled due to failure will be executed after any events initiated by the failed process with a guard that was satisfied prior to the failure. From this we see that if a failure occurs, the cleanup operation will not execute until any pending updates have completed.

A replicated database implemented in the above manner should perform quite well. Updates will be asynchronous [Joseph], and correctness and fault-tolerance will follow from the fact that the DB is causally consistent. In fact, good performance using these techniques was measured in our previous work, and reported in [Birman-b]. A weakness of the above implementation is that only one class of locks is supported, hence although reads are local there is no notion of a local read-lock. This limitation could certainly be overcome in a more sophisticated implementation.

9. Conclusions

We have proposed that advisory bulletin boards be considered as an alternative to more imperative styles of interaction in fault-tolerant distributed systems, and illustrated the approach with a series of examples that are straightforward when implemented as bboards and more complex when implemented using other programming methodologies. We do not view bboards as the only facility to be used in such systems, and indeed continue to believe that the mechanisms proposed in our previous work (resilient objects, fault-tolerant process groups) can play an important role. Rather, it is our feeling that if a diversity of fault-tolerant programming tools can be provided to distributed systems architects, then they will ultimately find it as easy to build fault-tolerant distributed software as it currently is to build fault-*intolerant* non-distributed software.

10. Acknowledgements

We are grateful to the members of the ANSA project for their many thoughtful comments.

11. References

[Birman-a] Birman, K. Replication and availability in the *ISIS* system. *10th ACM Symposium on Operating Systems Principles*, appearing as *Operating Systems Review* 19, 5 (Dec. 1985), 79-86.

[Birman-b] Birman, K., Joseph, T., Stephenson, P. Reliable communication in the presence of failures. Dept. of Computer Science, Cornell Univ., TR 86-694, August 1985 (revised August 1986). Accepted for publication: *ACM Transactions on Computer Systems.*

[Birrell] Birrell, A., Nelson, B. Implementing remote procedure calls. *ACM Transactions on Computer Systems* **2**, 1 (Feb. 1984), 39-59.

[Cheriton] Cheriton, D. Preliminary thoughts on problem oriented shared memory: a decentralized approach to distributed systems. *Operating systems review* **19**, 4 (Oct. 1985), 26-33.

[Carriero] Carriero, N., Gelernter, D. The S/Net's Linda Kernel. *10th ACM Symposium on Operating Systems Principles*, appearing as *Operating Systems Review* 19, 5 (Dec. 1985), 160.

[Chandy] Chandy, M., Lamport, L. Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems* **3**, 1 (Feb. 1985), 63-75.

[Chang] Chang, J, Maxemchuk, N. Reliable broadcast protocols. *ACM TOCS* **2**, 3 (Aug. 1984), 251-273.

[Cooper] Cooper, E. Replicated distributed programs. *10th ACM Symposium on Operating Systems Principles*, appearing as *Operating Systems Review* 19, 5 (Dec. 1985), 63-78.

[Cristian] Cristian, F., *et al.* Atomic broadcast: From simple message diffusion to Byzantine agreement. IBM Technical Report.

[Gray] Gray, J. Notes on database operating systems. *Lecture notes in computer science* 60, Goos and Hartmannis, *eds*. Springer Verlag 1978.

[Joseph] Joseph, T. and Birman, K. Low cost management of replicated data in fault-tolerant distributed systems. *ACM TOCS* 4, 1, Feb. 1986, 54-70.

[Koo] Koo, R., Toueg, S. Checkpointing and Rollback-Recovery for Distributed Systems. TR 85-706, Dept. of Computer Science, Cornell University (Oct. 1985).

[Lamport] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM* 21, 7, July 1978, 558-565.

[Peterson] Peterson, J. and Silbershatz, A. *Operating system concepts*, 2nd edition, Addison-Wesley Publishing Company, 1985.

[Schneider] Schneider, F., Gries, D., Schlicting, R. Reliable broadcast protocols. *Science of Computer Programming* 3, 2 (March 1984).

[Skeen] Skeen, D. Determining the last process to fail. *ACM TOCS* 3, 1, Feb. 1985, 15-30.